UNICORN ENGINE

WHAT, WHY, HOW.

Nicolò Boatto
Senior Security Consultant

IOActive

Unicorn Engine: What is it?

- Unicorn is a lightweight, multi-platform, multi-architecture CPU emulator framework, based on QEMU.
- Supports X86, ARM, ARM64 (ARMv8), M68K, MIPS, PowerPC, RISCV, SPARC, S390X, and TriCore.
- Implemented in C, with bindings for Crystal, Clojure, Visual Basic, Perl, Rust, Ruby, Python, Java, .NET, Go, Delphi/Free Pascal, Haskell, Pharo, and Lua.

IOActive

Unicorn Engine: Why?

- It can emulate pretty much anything.
- Allows emulation of arbitrary portions of binaries.
- It gives the ability to be extremely surgical with instruction-level hooking.
- Can save huge amounts of reversing time by just executing the parts we need and looking at the results.

Unicorn Engine: How?

- Example: Firmware Update Decryption
  - We have a decrypted firmware image from a device and an encrypted “firmware update” (it’s just a string)
  - 2d685cbae23ae7f71b97df2cb21d955f

IOActive

Unicorn Engine: How?

- Pop the firmware image into your RE tool of choice.
- Look for interesting functions.
- Build our emulation environment.
  - Run emulation.
  - Check results.
  - Make changes.
  - Repeat.
- Allows us to break down RE into individual problems.

main and do_upgrade

```c
int main(void)

{
  int iVar1;

  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_I2C1_Init();
  MX_I2S3_Init();
  MX_SPI1_Init();
  MX_USB_HOST_Init();
  MX_MBEDTLS_Init();
  iprintf("System Started...");
  iprintf("Looking for Updates...");
  iprintf("Updating Firmware...");
  iVar1 = do_upgrade();
  if (iVar1 == 0) {
    iprintf("Firmware Updated Successfully!");
  }
  do {
    MX_USB_HOST_Process();
    iprintf("Doing Firmwarey Stuff...");
  } while( true );
}
```

```c
int do_upgrade(void)

{
  uint32_t uVar1;
  uchar *firmware;
  char *pcVar2;
  uchar *decrypted;
  size_t in_size;
  int iVar3;
  char *decrypted_fw;
  uint32_t update_size;
  char *downloaded_fw;
  uint32_t size;

  iprintf("Downloading Firmware");
  uVar1 = get_fw_download_size();
  firmware = (uchar *)malloc(uVar1);
  pcVar2 = dummy_fw_download();
  if (pcVar2 == (char *)0x0) {
    uVar1 = get_fw_update_size((char *)firmware);
    decrypted = (uchar *)malloc(uVar1);
    in_size = strlen((char *)firmware);
    decrypt_firmware(firmware,in_size,decrypted,uVar1);
    iVar3 = 0;
  }
  else {
    iVar3 = -1;
  }
  return iVar3;
}
```


decrypt_firmware

```
int decrypt_firmware(char *encrypted_firmware,uint32_t in_size,char *decrypted_firmware,
                    uint32_t out_size)

{
  mbedtls_aes_context *ctx;
  int iVar1;
  uchar iv [16];
  uchar key [32];
  mbedtls_aes_context *aes_context;

  ctx = (mbedtls_aes_context *)malloc(0x118);
  if ((*encrypted_firmware == '\0') || (in_size == 0)) {
    iVar1 = -1;
  }
  else {
    get_firmware_iv(iv);
    get_firmware_key(key);
    mbedtls_aes_init(ctx);
    mbedtls_aes_setkey_dec(ctx,key,0x100);
    mbedtls_aes_crypt_cbc(ctx,0,in_size,iv,(uchar *)encrypted_firmware,(uchar *)decrypted_firmware);
    free(ctx);
    iVar1 = 0;
  }
  return iVar1;
}
```

get_firmware_iv

```c
int get_firmware_iv(uchar *iv)

{
  uchar sha_output [32];
  uchar secret_input [16];

  secret_input._0_4_ = 0x52544f4e;
  secret_input._4_4_ = 0x4c4c4145;
  secret_input._8_4_ = 0x43455359;
  secret_input._12_4_ = 0x544552;
  mbedtls_sha256(secret_input,0x10,sha_output,0);
  memcpy(iv,sha_output,0x10);
  return 0;
}
```


IOActive

get_firmware_key

```
int get_firmware_key(uchar *key)

{
  char encrypted_key [32];
  char super_secret_kek [26];

  super_secret_kek._0_4_ = 0x74616877;
  super_secret_kek._4_4_ = 0x65707573;
  super_secret_kek._8_4_ = 0x63657372;
  super_secret_kek._12_4_ = 0x6b746572;
  super_secret_kek._16_4_ = 0x73697965;
  super_secret_kek._20_4_ = 0x73696874;
  super_secret_kek._24_2_ = 0x3f;
  encrypted_key._0_4_ = 0x44332211;
  encrypted_key._4_4_ = 0x88776655;
  encrypted_key._8_4_ = 0xccbbaa99;
  encrypted_key._12_4_ = 0x11ffeedd;
  encrypted_key._16_4_ = 0x44443322;
  encrypted_key._20_4_ = 0x88776655;
  encrypted_key._24_4_ = 0xccbbaa99;
  encrypted_key._28_4_ = 0xffeedd;
  super_secret_encryption(super_secret_kek,encrypted_key,key);
  return 0;
}
```

super_secret_encryption

```c
int super_secret_encryption(char *key,char *plaintext,uchar *ciphertext)

{
  uchar S [256];

  ksa(key,S);
  prga(S,plaintext,ciphertext);
  return 0;
}
```

```c
int ksa(char *key,uchar *S)

{
  size_t sVar1;
  uint uVar2;
  int len;
  int i_1;
  int i;
  int j;

  sVar1 = strlen(key);
  j = 0;
  for (i = 0; i < 0x100; i = i + 1) {
    S[i] = (uchar)i;
  }
  for (i_1 = 0; i_1 < 0x100; i_1 = i_1 + 1) {
    uVar2 = (uint)(byte)key[i_1 - sVar1 * ((uint)i_1 / sVar1)] + (uint)S[i_1] + j;
    j = uVar2 & 0xff;
    if ((int)uVar2 < 1) {
      j = -(-uVar2 & 0xff);
    }
    swap(S + i_1,S + j);
  }
  return 0;
}
```

```c
int prga(uchar *S,char *plaintext,uchar *ciphertext)

{
  size_t sVar1;
  uint uVar2;
  int rnd;
  size_t len;
  size_t n;
  int j;
  int i;

  i = 0;
  j = 0;
  n = 0;
  sVar1 = strlen(plaintext);
  for (; n < sVar1; n = n + 1) {
    uVar2 = i + 1;
    i = uVar2 & 0xff;
    if ((int)uVar2 < 1) {
      i = -(-uVar2 & 0xff);
    }
    uVar2 = j + (uint)S[i];
    j = uVar2 & 0xff;
    if ((int)uVar2 < 1) {
      j = -(-uVar2 & 0xff);
    }
    swap(S + i,S + j);
    ciphertext[n] = S[(byte)(S[j] + S[i])] ^ plaintext[n];
  }
  return 0;
}
```

Let’s do this the easy way.

```
int decrypt_firmware(char *encrypted_firmware,uint32_t in_size,char *decrypted_firmware,
                    uint32_t out_size)

{
  mbedtls_aes_context *ctx;
  int iVar1;
  uchar iv [16];
  uchar key [32];
  mbedtls_aes_context *aes_context;

  ctx = (mbedtls_aes_context *)malloc(0x118);
  if ((*encrypted_firmware == '\0') || (in_size == 0)) {
    iVar1 = -1;
  }
  else {
    get_firmware_iv(iv);
    get_firmware_key(key);
    mbedtls_aes_init(ctx);
    mbedtls_aes_setkey_dec(ctx,key,0x100);
    mbedtls_aes_crypt_cbc(ctx,0,in_size,iv,(uchar *)encrypted_firmware,(uchar *)decrypted_firmware);
    free(ctx);
    iVar1 = 0;
  }
  return iVar1;
}
```

load_elf

```python
def main():
    if trace_file:
        sys.stdout = open('./trace.txt','w')
    with open("./UnicornTarget.elf", "rb") as elf:

        # Address of the decrypt_firmware function that we're attacking
        decrypt_firmware = 0x8000585

        # Set the emulation start address as decrypt_firmware
        #   and end once we have what we need.
        emu_start = decrypt_firmware
        emu_end = 0x80005f4

        print("Initialising Emulator...")
        mu = init_emulation(elf)
```

init_emulation

```python
def init_emulation(elf: BufferedReader):
    global lief_binary

    mu = Uc(UC_ARCH_ARM, UC_MODE_MCLASS)

    # Map normal working regions
    mu.mem_map(STACK_BASE, STACK_SIZE)
    mu.mem_map(HEAP_BASE, HEAP_SIZE)

    initial_sp = STACK_BASE + STACK_SIZE
    mu.reg_write(UC_ARM_REG_SP, initial_sp)

    lief_binary = lief.parse(elf)

    build_symbol_table(lief_binary)

    # Offsets of the the .isr_vector, .text, and .rodata sections of the elf
    isr_vector = lief_binary.get_section(".isr_vector")
    ISR_OFFSET      = isr_vector.file_offset
    ISR_SIZE        = isr_vector.size
    ISR_ADDRESS     = isr_vector.virtual_address

    text = lief_binary.get_section(".text")
    TEXT_OFFSET     = text.file_offset
    TEXT_SIZE       = text.size
    TEXT_ADDRESS    = text.virtual_address

    rodata = lief_binary.get_section(".rodata")
    RODATA_OFFSET   = rodata.file_offset
    RODATA_SIZE     = rodata.size
    RODATA_ADDRESS  = rodata.virtual_address

    TOTAL_SIZE = RODATA_OFFSET + RODATA_SIZE - ISR_OFFSET

    # Map a single memory block that allows us to get around the Unicorn engine mapping limitations
    mu.mem_map(ELF_BASE, resize_to_block(TOTAL_SIZE, BLOCK_SIZE))

    elf.seek(ISR_OFFSET)
    mu.mem_write(ISR_ADDRESS, elf.read(ISR_SIZE))

    elf.seek(TEXT_OFFSET)
    mu.mem_write(TEXT_ADDRESS, elf.read(TEXT_SIZE))

    elf.seek(RODATA_OFFSET)
    mu.mem_write(RODATA_ADDRESS, elf.read(RODATA_SIZE))

    return mu
```

decrypt_firmware

```c
int decrypt_firmware(char *encrypted_firmware,uint32_t in_size,char *decrypted_firmware,
                    uint32_t out_size)

{
  mbedtls_aes_context *ctx;
  int iVar1;
  uchar iv [16];
  uchar key [32];
  mbedtls_aes_context *aes_context;

  ctx = (mbedtls_aes_context *)malloc(0x118);
  if ((*encrypted_firmware == '\0') || (in_size == 0)) {
    iVar1 = -1;
  }
  else {
    get_firmware_iv(iv);
    get_firmware_key(key);
    mbedtls_aes_init(ctx);
    mbedtls_aes_setkey_dec(ctx,key,0x100);
    mbedtls_aes_crypt_cbc(ctx,0,in_size,iv,(uchar *)encrypted_firmware,(uchar *)decrypted_firmware);
    free(ctx);
    iVar1 = 0;
  }
  return iVar1;
}
```

hook_code

```python
def hook_code(mu : unicorn.Uc, address, size, data):
    global last_function
    global trace
    global full_trace
    global lief_binary
    global symbol_table
    function_name = get_function_name(address, symbol_table)

    # Print function name and register values each time a new function is called
    if ((function_name != last_function) and trace == 1 ) or full_trace == 1:
        print(">>> Tracing instruction at 0x%x, in function %s" % (address, function_name))
        print_regs(mu)
    last_function = function_name

    # FUNCTION SUBSTITUTION
    # Substitute my own implementation of malloc to be able to run in a limited environment
    if address == lief_binary.get_symbol("malloc").value + 1:
        malloc(mu)
    # Substitute my own implementation of SHA256 to skip the emulated one
    elif address == lief_binary.get_symbol("mbedtls_sha256").value + 1:
        SHA256(mu)
    # Substitute my own implementation of setkey to skip the emulated one and intercept the key
    elif address == lief_binary.get_symbol("mbedtls_aes_setkey_dec").value + 1:
        mbed_setkey(mu)
    # Substitute my own implementation of setkey to skip the emulated one and intercept the iv
    elif address == lief_binary.get_symbol("mbedtls_aes_crypt_cbc").value + 1:
        mbed_aes_crypt(mu)
```


trace_file

```
>>> Tracing instruction at 0x8000584, in function decrypt_firmware
SP = 0x3002000
PC = 0x8000584
R0 = 0x9000000
R1 = 0x10
R2 = 0x0
R3 = 0x0
R4 = 0x0
>>> Tracing instruction at 0x800b284, in function malloc
SP = 0x3001fa8
PC = 0x800b284
R0 = 0x118
R1 = 0x10
R2 = 0x0
R3 = 0x0
R4 = 0x0
>>> Tracing instruction at 0x800059a, in function decrypt_firmware
SP = 0x3001fa8
PC = 0x800059a
R0 = 0x9000010
R1 = 0x10
R2 = 0x0
R3 = 0x20000030
R4 = 0x0
```

function_substitutions

```python
# Substitute SHA256 function, which just uses the python version
#   The original implementation caused issues and crashed the emulation
def SHA256(mu : unicorn.Uc):
    input_address  = mu.reg_read(UC_ARM_REG_R0)
    input_length   = mu.reg_read(UC_ARM_REG_R1)
    input = mu.mem_read(input_address, input_length)
    hash = hashlib.sha256(input).digest()
    print("SHA256 Input:  %s" % input.decode("utf8"))
    print("SHA256 Output: %s" % bytes(hash).hex())

    output_address = mu.reg_read(UC_ARM_REG_R2)
    mu.mem_write(output_address, hash)

    # Return to the end of the SHA256 function
    mu.reg_write(UC_ARM_REG_PC, mu.reg_read(UC_ARM_REG_LR))
```

```python
# My own malloc implementation, which just stores things sequentially
def malloc(mu : unicorn.Uc):
    global heap_pointer

    size = mu.reg_read(UC_ARM_REG_R0)
    address = internal_malloc(mu, size)
    mu.reg_write(UC_ARM_REG_R0, address)

    # Restore execution to the end of the malloc function
    mu.reg_write(UC_ARM_REG_PC, mu.reg_read(UC_ARM_REG_LR))


def internal_malloc(mu: unicorn.Uc, size):
    global heap_pointer

    address = heap_pointer
    print("Malloc called with size: %x" % size)
    heap_pointer = heap_pointer + size
    return address
```


more_function_substitutions

```python
# Substitute the setkey function with one that just prints the key
def mbed_setkey(mu: unicorn.Uc):
    global key

    key_address = mu.reg_read(UC_ARM_REG_R1)
    key = mu.mem_read(key_address, 32)
    print("Key = %s" % key.hex())
    # Return to the end of the setkey function
    mu.reg_write(UC_ARM_REG_PC, mu.reg_read(UC_ARM_REG_LR))
```

```python
# Substitute the crypt function with one that just prints the iv
def mbed_aes_crypt(mu: unicorn.Uc):
    global iv

    iv_address = mu.reg_read(UC_ARM_REG_R3)
    iv = mu.mem_read(iv_address, 16)
    print("IV = %s" % iv.hex())
    # Return to the end of the crypt function
    mu.reg_write(UC_ARM_REG_PC, mu.reg_read(UC_ARM_REG_LR))
```


IOActive

emu_start

```
# All application flags turned on
        mu.reg_write(UC_ARM_REG_APSR, 0xFFFFFFFF)

        # Set thumb mode in the CPSR
        mu.reg_write(UC_ARM_REG_CPSR, 0x20)

        # Write the "firmware" into the heap
        firmware = bytes.fromhex("2d685cbae23ae7f71b97df2cb21d955f")
        firmware_size = len(firmware)
        firmware_address = internal_malloc(mu, firmware_size)
        mu.mem_write(firmware_address, firmware)

        # Set up register state for decrypt_firmware
        mu.reg_write(UC_ARM_REG_R0, firmware_address)
        mu.reg_write(UC_ARM_REG_R1, firmware_size)
        mu.reg_write(UC_ARM_REG_R2, 0)
        mu.reg_write(UC_ARM_REG_R3, 0)

        try:
            # Starting emulation
            mu.reg_write(UC_ARM_REG_PC, emu_start)
            mu.hook_add(UC_HOOK_CODE, hook_code)
            print("Emulation started.")
            mu.emu_start(emu_start, emu_end)
            print("Emulation ended, decrypting secret...")
            print(decrypt(firmware))
```

demo

no_demo?

```
❯ /bin/python3 /home/b/Desktop/UnicornTalk/UnicornTalkFWDecrypt/emulate_fw_decrypt.py
Initialising Emulator...
Building Symbol Table...
Malloc called with size: 10
Emulation started.
Malloc called with size: 118
SHA256 Input:  NOTREALLYSECRET
SHA256 Output: 4c58f08235928c9b7c728a77c6bb8ba02c110ec4a91607eed3e632818e329547
Key = b79ddc557d6eff8b30b4aa57dc185147d6bb0d3869817c5a0207b7b029bd4e00
IV = 4c58f08235928c9b7c728a77c6bb8ba0
Emulation ended, decrypting secret...
UNICORN FTW!
```

questions?